








EX LIBRIS  
UNIVERSITATIS  
ALBERTENSIS

---

The Bruce Peel  
Special Collections  
Library







Digitized by the Internet Archive  
in 2025 with funding from  
University of Alberta Library

<https://archive.org/details/016201264240>





**University of Alberta**

**Library Release Form**

**Name of Author:** Neil Edward Burch

**Title of Thesis:** Self Stabilisation and Analog Computation

**Degree:** Master of Science

**Year this Degree Granted:** 2001

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.





University of Alberta

SELF STABILISATION AND ANALOG COMPUTATION

by

Neil Edward Burch



A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta  
Fall 2001





University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Self Stabilisation and Analog Computation** submitted by Neil Edward Burch in partial fulfillment of the requirements for the degree of **Master of Science**.





# Abstract

Self stabilising systems, where a target state is reached regardless of the initial state, are one of the key notions in both distributed computing systems and dynamical systems theory. In the computing world, such systems consist of communicating finite-state automata. Of particular interest are “uniform” systems in which all of the processors are identical, with no distinguishing features. In such cases, randomness is required to break symmetry in order for the system to stabilise.

At the fundamental level, discrete computing devices are built from devices with continuous behaviour. This thesis investigates how the concepts of discrete self stabilising systems transfer over to self stabilising systems of analog machines, which have continuous real valued states.

The first requirement was to come up with a model of a system of continuous state machines. We chose to use analog machines constructed from operational amplifiers as the basis for our model. The second part of the research was to create and examine some uniform self stabilising systems using this model. We give a protocol for orienting a ring, one of the canonical problems in self stabilisation, and then prove its correctness. We also give details about the hardware that we used for experimentation and implementing protocols.





# Contents

<b>1</b>	<b>Self Stabilisation and Analog Computation</b>	<b>1</b>
<b>2</b>	<b>History</b>	<b>4</b>
<b>3</b>	<b>Model of Computation</b>	<b>10</b>
<b>4</b>	<b>Self Stabilisation</b>	<b>14</b>
<b>5</b>	<b>Ring Orientation</b>	<b>17</b>
<b>6</b>	<b>Correctness of Protocol</b>	<b>21</b>
6.1	Overview . . . . .	22
6.2	Reachability of Target . . . . .	23
6.3	Proof of Stabilisation . . . . .	32
<b>7</b>	<b>Hardware Implementation</b>	<b>34</b>
<b>8</b>	<b>Average Time, Better Bounds, and Simulation</b>	<b>39</b>
<b>9</b>	<b>Conclusions</b>	<b>43</b>
	<b>Bibliography</b>	<b>45</b>





# List of Tables

7.1	processor one forced to be negative . . . . .	37
7.2	processor one forced to be negative with a pulse . . . . .	38
8.1	transition function for discrete processor . . . . .	40
8.2	average number of steps to stabilise . . . . .	42



# List of Figures

2.1	Hardware approximation of the sign function. . . . .	7
5.1	A ring of four processors. . . . .	19
6.1	A chain of processors. . . . .	22
6.2	Case 1. . . . .	25
6.3	Case 2. . . . .	25
6.4	Case 3 - special configurations. . . . .	26
6.5	Three cases for a facing pair of processors. . . . .	26
6.6	$x$ has reached zero before $y$ . . . . .	28
6.7	Case 3 - alternating special configurations cancel movement of weakly oriented processors. . . . .	28
7.1	Circuit diagram for the ring orientation protocol . . . . .	35
8.1	average number of steps to stabilise . . . . .	42





# Chapter 1

## Self Stabilisation and Analog Computation

Computing science and computation is generally associated with a digital world where discrete information is read and processed at discrete time intervals. This is not surprising: the digital computer has had a large impact on the world. As a piece of physical hardware, however, at some level its operation must be as continuous as any other physical processes. This universality is one reason analog computation is interesting.

We chose to do two things with analog computation. The first was to come up with a model of analog computation suitable for work with distributed continuous computation, as with a collection of analog computers. The other was to implement some simple algorithms in hardware and use the model to prove correctness.

The analog computation that we are interested in is continuous in both time and state. That is to say, the state of a variable is a real number and this value is a continuous function of time. Computation is performed by having the derivative of this function depend on the value of that variable and others.

An example of this would be a machine with two variables  $d$  and  $v$ , where  $d = 0$  and  $v = 0$  at time  $t = 0$ ,  $\frac{\delta d}{\delta t} = v$ , and  $\frac{\delta v}{\delta t} = 9.8$ . Then at time  $t$ ,  $d = \frac{9.8t^2}{2}$ . This is approximately the distance that a free falling body would





fall in  $t$  seconds on Earth, starting from rest.

In a distributed system there is more than one machine, with each machine having access to some of the values of other machines. For example two systems  $A$  and  $B$  have variables  $x_A$  and  $x_B$  respectively, and  $\frac{\delta x_A}{\delta t} = \text{sign}(-x_B) - x_A$  and  $\frac{\delta x_B}{\delta t} = \text{sign}(-x_A) - x_B$ . In this system, putting aside the issue of what happens when both  $x_A$  and  $x_B$  are 0, each machine moves towards having a magnitude of one and a different sign than the other machine.

Looking back at the example that computes the distance an object falls, notice how getting the right answer depends on the initial values of the variables. If  $d$  or  $v$  are not 0, then the result is the answer to a different question. In contrast, a self stabilising system will eventually reach its destination state regardless of its initial state.

As an example, consider the problem of ring orientation. In this problem there are a number of processors, each with two neighbours, arranged in a circle. This is a ring. All of the processors run the same program, and no processor has an identifier that can be used to identify it. Further, while a processor can distinguish between its two neighbours, there is no specific ordering of the neighbours. This means, for example, that it is not possible to trace a path around the circle by always going from a processor to its first neighbour. The problem is to set the state of all the processors so that each has an arrow pointing to one of its neighbours, and a path around the circle can be traced by following these arrows.

A collection of programs that run on a collection of processors to solve a single problem is sometimes called a protocol. So, a simple self stabilising protocol for solving this problem on discrete machines is as follows. Each processor has one variable, which can have either the value 1 or 2. This is the arrow: if the value is 1 the processor is pointing to its first neighbour, otherwise it is pointing to its second neighbour. For every processor  $p$ , at each



update,  $p$  looks at the neighbour it is pointing at. If this neighbour is pointing at  $p$ ,  $p$  will choose a random direction for its arrow.

If all processors point in the same direction  $D$  around the ring, the system is oriented, and will stay that way. If not, randomly choose a direction around the ring. After some number of processor activations  $t$ , all processors must have activated. At some of these processor activations nothing happens. In the others, there is a 50% chance that the processor will now point in the chosen direction  $D$ . So with probability greater than  $\frac{1}{2}^t$ , every processor that could flip now points in direction  $D$ . This means that at least one more processor points in direction  $D$ . Thus if there are  $m$  processors that are not facing in direction  $D$ , the probability that all processors will point in direction  $D$  in  $m$  steps is at least  $(\frac{1}{2})^m$ . This is small, but non-zero, so as time goes on, the probability that the system will become oriented approaches 1. Note that there is no required initial state. The system just heads towards the final state.

In the following chapters we formalise these concepts and discuss them in more detail. In chapter 2 we give some background on how we got to where we are now. In chapters 3 and 4 we formally discuss the model of computation and self stabilisation under this model. Chapters 5 and 6 define a solution for ring orientation and prove its correctness. We talk about the hardware we used in chapter 7. Chapter 8 gives the work we did to find the average time to stabilise for a system running the ring orientation protocol. Finally, we give our conclusions in chapter 9.





# Chapter 2

## History

This chapter is an overview of the progression that our work took. Starting with a pre-existing model, this chapter goes through in rough chronological order the ideas for models and protocols. Each of these ideas, and the problems that arose from them, contributed in some way to the model and protocol we present in later chapters.

In [4], which inspired this work, the model of computation that is informally proposed is an extension of an another existing model, arithmetic circuits. The modifications made involved adding cycles with integration gates to arithmetic circuits for the storing of state, and noise to generate randomness for such tasks as symmetry breaking. A valid system in this model is a graph with the arithmetic operators  $+$ ,  $-$ ,  $x$ ,  $^{-1}$ , and integration at the vertices, with at least one integration gate in every cycle. A major factor in the design of this model was efficient (few gates) approximation and efficient (polynomial time complexity) simulation of non-linear functions.

While this model is simple, and is built upon well studied work, it can be hard to work with, and is not a completely specified. In theory, there is no need for discontinuous functions in a model of analog computation because they do not exist in the hardware, and they can be approximated to any desired degree of accuracy. However, in practice, systems often have to do decision making,



and an approximation of some discontinuous function is used to do this. For example, you may want a variable to increase if variable  $x$  is positive, and stay the same if it is negative. This can be done with the discontinuous function  $\text{sign}(x) + 1$ . Because there are no built in discontinuous functions in this model, each function like this must be individually approximated.

Another difficulty is that solving problems involves talking about a system of differential equations. These equations are unlikely to be pleasant. Something must be done to ensure the values are bounded at all times, there may be arbitrarily many equations if there are arbitrarily many processors or variables being examined, and the functions are arbitrary, within the limitations that they are composed of  $+$ ,  $-$ ,  $x$ ,  $x^{-1}$ , and integration. Working with such a system can be difficult.

In the same paper, there are protocols given for three basic self stabilisation problems: pair distinguishing, ring orientation, and token passing. While we originally intended to look at all three of these problems, it became clear that this project would be too large. Of the three problems, we chose ring orientation to work with. Pair distinguishing was too small, being limited to two processors, and token passing, as the most complicated system of the three, seemed a poor choice for our first problem.

The protocol given for ring orientation behaves as follows. Each processor has an arrow which points towards a “left” or “right” neighbour, or up, at neither neighbour. Left and right is an arbitrary choice of every processor, and does not correspond to a global idea of left and right. If the neighbour which a processor  $p$  is facing points back at  $p$ , then processor  $p$  should randomly flip its arrow. The implementation uses two variables. One of the variables of each processor is interpreted as the arrow pointing towards one of its neighbours, with negative being towards one neighbour, and positive towards the other neighbour. The other variable represents the current rate of change of the





arrow. Note how this is similar to the simple discrete protocol, except that now the position of the arrow can vary smoothly between left and right, and the processor remembers the speed with which the arrow is changing.

The first cut at a formal model was a slightly modified version of the model in [4]. The major difference was a conceptual change. Instead of considering each processor to be a graph with operators at each vertex, we considered each variable to be the integration of some function. While mathematically identical, this model makes an explicit distinction between variables and the function which controls it, while ignoring the structure of the function.

Because of the similarity to the model it is based on, this model has the same problems. Discontinuous functions must still be approximated by the designer, and the underlying math is still difficult to work with.

The first attempt at a new protocol for ring orientation was more clearly different. Instead of using two variables to represent the direction of an arrow and its speed, we tried making a protocol using only a single variable representing the direction of an arrow. The underlying idea is otherwise the same. There is an arrow pointing somewhere between a left and a right neighbour, and any processor that faces a processor pointing back at it randomly flips its arrow.

While the protocol appeared to work, both on a ring of four actual hardware machines and a software simulation of sixteen processors, no real effort was made yet to verify correctness. Also, even though the function for this protocol used fewer gates than the original protocol, it was still large enough that we only had sufficient hardware to build four machines.

While trying to make the transition functions simple enough to make building a larger hardware version possible, we thought of two things. The first was that multiplying a value by a large enough constant provided an excellent approximation of the sign function. The second was that the magnitude of the



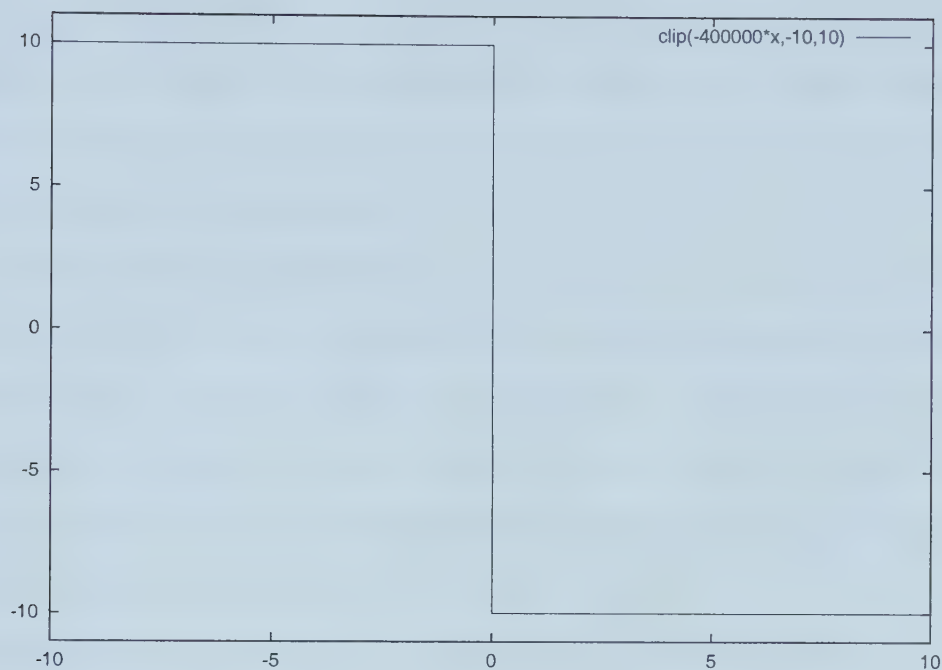


Figure 2.1: Hardware approximation of the sign function.

transition function was irrelevant, and only the sign mattered.

The approximation of the sign function came about as follows. The operational amplifier we used, the AD713, has a typical open-loop gain of roughly 400000. What this meant to us was that if we simply fed the signal into the op-amp without feedback, we would get a greatly magnified signal, albeit a negated one. Because the hardware has a maximum magnitude for output signals, this meant that except for a very small region around 0, the output would have the maximum magnitude. Working with the values of  $\pm 10$  volts that we used, this gives the graph in 2.1. While this must be scaled again to produce the sign function, this is just a simple matter of scaling by a negative constant, which is easy to do with the hardware.

This is an excellent approximation of the sign function. It is made more appealing by noting that region where the approximation breaks down behaves nicely (the function is, after all, a straight line) and has a smaller range than



the noise. The last point is important because this means that noise helps mask the region where the approximation breaks down. These properties made it seem reasonable to add the sign function to the model, so that complex approximations were not needed.

Because in all of the problems we were looking at, it also appeared that the magnitude was irrelevant, it seemed like we could just divide the transition function into a number of cases, with the function being either *maxval* or  $-maxval$  in each case. In order to get a variable to stay at a certain value, the value must be along a boundary between two cases which both push the variable towards the value.

The protocol we built upon this model was a refinement of the previous protocol. The transition function for the variable had the same four conceptual regions, as before (neighbours facing towards, facing away, facing left, and facing right.) and the direction of change was the same. Now, however, the regions were exactly defined within the transition function using the sign function, and the magnitude of the function within a region was the same throughout the region.

This model was revised fairly quickly. In a system running the protocol briefly described above, consider a processor  $p$  whose left neighbour points straight up, and whose right neighbour points to  $p$ . Due to noise, the left neighbour will spend roughly half of its time pointing towards  $p$ , and half its time pointing away from  $p$ . This means half the time  $p$  will move up, and half the time it will move left. Since it moves at the same rate in both cases,  $p$  will remain roughly stationary. The original intent was for  $p$  to move upwards, and there did not appear to be an easy way to do this without allowing values between *maxval* and  $-maxval$ .

The first solution we considered was to still divide the domain of the function into homogeneous regions, but allow any value within the region, not just





*maxval* and  $-maxval$ . While this works for ring orientation, we could come up with no compelling argument for why this would be sufficient in all cases.

While the idea of only using homogeneous regions is appealing, the lack of an argument about the expressiveness about this model meant this idea had to be dropped. Instead, we added the ability (as opposed to the requirement) to have regions by allowing the transition function to be defined as multiple continuous pieces (a piecewise function.) This gave the current model, described in chapter 3

Along with this model change, there were three changes to the protocol for ring orientation. The first, most obvious change, was to place the function in the correct format. That is, it was defined as a piecewise function. The second change was to make the processor change more quickly when both its neighbours face it than in any other case. This means that a processor with a neighbour that points up will head up as well. The final change was that in the case where both neighbours point away from a processor, the processor will not change. This was done because it made arguing correctness easier, and did not seem to otherwise affect the performance.



# Chapter 3

## Model of Computation

In this chapter we define the model that we intend to use. We define single processors and how they change state, and then define how a system of processors is put together. Finally, we define some terminology that will be used in later discussion.

The proposed model for distributed analog computation is based on collections of analog machines constructed from operational amplifiers and analog multipliers. It can compute the same kind of functions that the hardware can, and includes random noise and something analogous to processor scheduling in discrete systems. It does not take into account factors like propagation delay. It also does not allow for an infinite number of variables or processors.

The model, then, is as follows. A system consists of a set of processors with unique identifiers, a set of pairs of identifiers that list which processors are adjacent (neighbours), and the global constant  $maxval$ . The identifiers are not accessible to any processor if the processors are anonymous. If all the processors are identical except for their identifiers, the system is said to be *uniform*.

Each processor has a number of state variables, each of which holds one real value which is bounded by  $\pm maxval$  and has a transition function that determines its instantaneous rate of change. There are also pseudo variables,





which are functions of the processor's state variables. The value of pseudo variables are only exported to other processors. The processor also has access to an ordered set of its neighbours identifiers. For each of the processors in this set, there is an ordered subset of the variables and pseudo variables which determines the values the processor exports to its neighbours. The order of these sets is determined by the designer.

The system also has random noise. The noise modeled corresponds to thermal noise within the hardware. This is done by adding a global constant *maxnoise* and an independent random noise function to each variable of every processor. The magnitude of the noise is always strictly less than *maxnoise* and must be able to change more rapidly than any variable. To distinguish between the value of a processor with and without noise the terms stored and net value are introduced. The value of a processor's variable without noise is the *stored* value. The *net* value of a variable is the stored value of the variable, plus the current value of the random noise applied to that variable. Everything, including external observers and other processors, refers to the net value of the variable.

The transition function of a variable is a function of any number of the net values of the processor's own variables and the values exported by the processor's neighbours. This function gives the derivative of the variable with respect to time. For example, if a variable  $V$  has transition function  $f(V)$  and random noise  $r(T)$ , the function  $V(T)$  describing the stored value over time is the solution to the differential equation  $\frac{dV(T)}{dT} = f(V(T) + r(T))$  that has  $V(0)$  equal to the initial value of  $V$ .

The transition function can be piecewise defined along its domain. Each piece of the function must be composed of constants and the variables listed above using addition, multiplication, subtraction, division. The value of this function, like the values of the variables, is limited in magnitude by  $\pm maxval$ .



While analog hardware can, in general, only deal with continuous functions, a piecewise defined functions is a reasonable transition function because the sign function can be reasonably approximated in hardware. Consider the line  $y = mx$  computed in hardware. Because there is a maximum value the hardware can have, for any values of  $x > \text{maxval}/m$ ,  $y = \text{maxval}$ , with similar behaviour for negative values. So outside the interval  $(-\text{maxval}/m, \text{maxval}/m)$ ,  $y = mx$  behaves like the sign function. Given a large enough  $m$ ,  $\text{maxval}/m$  can be made arbitrarily small. A piecewise function is simply the sum of the products of shifted sign functions and the functions that make up the piecewise function.

The functions defining the pseudo variables have the same limitations as the transition functions for variables, except that they can only depend upon the net values of the processor's own variables. An example would be a pseudo variable which is the negation of a particular variable.

This definition of the transition function lends itself well to the following measure of size complexity. Let the size of a function be the number of operators it has, plus the number of pieces in the function minus one. Then the *size* of a machine is the sum of the sizes of the transition functions of each of its variables.

The instantaneous state of a processor with  $m$  state variables is a  $2m$ -tuple containing for each state variable both the stored value of the variable and the value of the noise for that variable at that instant in time. The state of a system with  $n$  processors is an  $n$ -tuple with each element of the tuple being a processor state tuple.

Conceptually, a *path* from state  $S_0$  to  $S_1$  exists in a system if there exists some values such that the system in state  $S_0$  ends up in  $S_1$  if the random noise takes these values. Let  $S(t)$  be the function that gives the state of a system at time  $t$ . Let  $v_{i,j}(t)$  be the stored value of processor  $i$ 's  $j$ th variable at time  $t$ ,



and  $r_{i,j}(t)$  the value of the random noise applied to it.  $v$  and  $r$  are projections of  $S$ . Let  $\delta_{i,j}(x)$  be the transition function applied to this variable. Note that if  $S$  describes the normal operation of a system, the derivative of  $v_{i,j}(t)$  with respect to  $t$  is equal to the value of the transition function applied to the net value of that variable (ie  $\delta_{i,j}(v_{i,j}(t) + r_{i,j}(t))$ ) for all  $t$ . A path exists from  $S_0$  to  $S_1$  if  $S(t_0) = S_0$  and there exists time  $t_1$  and functions  $r_{i,j}(t)$  such that the resultant  $S$  describes the normal operation of a system and  $S(t_1) = S_1$ .

Another concept needed is the idea of being close to a value. A variable is said to be close to a value  $v$  if the stored value of the variable is less than *maxnoise* away from  $v$ . Another way to say this would be that the stored value of the variable is in an open ball of radius *maxnoise* centered around  $v$ . This is used because with noise the variables will never stay at one exact value. So when considering system state it is usually necessary to consider whether a variable is close to a value, not whether it has exactly that value.

This definition can be extended to processors and systems. A processor is said to be close to a processor state if each variable is close to the value specified for that variable within the state. A system is said to be close to a state if each processor is close to the state specified for the processor.





# Chapter 4

## Self Stabilisation

We will now give definitions for self stabilisation in the context of the model we have defined. While most of the concepts remain mostly unchanged from those used in discussing discrete systems, the scheduling demon has no direct counterpart in continuous systems because the processor is always running. We give an alternative involving speeding up or slowing down processors.

A distributed system is said to be self stabilising if, regardless of the current state, it is expected to reach a target state and stay at that state. Target states are states specified by the designer for some reason (eg a final solution to a problem.) There may be more than one target state. These definitions apply to both discrete and continuous systems.

More precisely, in a continuous system, a state  $S$  is stable if for every system that is close to  $S$ , there does not exist a path to a state that is not close to  $S$ . That is, if the system is close to a stable state  $S$ , it will remain close to  $S$ . A system is self stabilising if all stable states are target states, and all states are expected to reach a state that is close to a stable state.

There is an additional complication. A discrete system is actually self stabilising with respect to a scheduler daemon. This models the interleaving of execution that can happen in a parallel system. The scheduling daemon schedules the execution of the processors so as to maximally hinder the progress



of the protocol running on the system. The scheduler is generally allowed to look at the state of processors, but not any future random values.

Different scheduling daemons have a differing amount of control over the execution. For example, a distributed daemon allows a processor to read from another processor and write back a new value in one atomic operation. Under a read/write daemon, a processor may only read or write a value from another processor without the possibility of interruption.

Because the scheduler daemon is an optimal adversary within its limits, if the protocol works under that scheduler, it will work under any normal operation with conditions that satisfy the same limitations as the daemon.

A similar concept is needed in analog systems. While processors are always changing, so that there can be no interleaving of processor execution, the rate that a processor changes given exactly the same set of inputs can vary from time to time and from processor to processor.

The way this is modeled is by allowing the scheduler to slow down individual processors by a finite amount. While this could be done by actually having different time scales for different processors, the same result can be achieved by simply scaling the output of the transition functions. So at every instant, for every processor, the scheduler multiplies the rate of change computed by the transition function for every variable of that processor by a value between 1 and  $\frac{1}{maxspeedup}$ , where *maxspeedup* is another global constant that determines how much the scheduler can affect processor speed.

This means that the derivative of the stored value of a variable is not just the transition function for the variable, but the transition function multiplied by scheduler speedup function for that variable's processor. A system state function  $S(t)$  describes the normal operation of a system if the derivative of  $v_{i,j}(t)$  with respect to  $t$  is equal to  $sched_i(t) * \delta_{i,j}(v_{i,j}(t) + r_{i,j}(t))$  for all  $t$ , where  $sched_i(t)$  describes the scaling factor the scheduler applies to processor  $i$  at



time  $t$ .

The presence of the scheduling daemon also changes what it means for there to exist a path from  $S_0$  to a state  $S_1$ . Now, instead of a path existing if there is a set of values for random noise that drive the system towards  $S_1$ , a path exists if  $S(t_0) = S_0$  and for every possible set of scheduler functions  $sched_i(t)$ , there exists time  $t_1$  and noise functions  $r_{i,j}(t)$  such that the resultant  $S$  describes the normal operation of a system and  $S(t_1) = S_1$ . The idea behind this is that a path is independent of the scheduler, so if a path to state  $S_1$  exists, the system can reach state  $S_1$  no matter what the scheduler does.





# Chapter 5

## Ring Orientation

In the previous chapters, we defined the model we are using and what it means to be self stabilising within this model. In this chapter, a self stabilising solution for the problem of orienting a ring will be given. A few properties of this system that will be useful in proving correctness are highlighted at the end of the chapter.

A ring is a system of processors arranged in a circle: each processor has exactly two unique neighbours. This does not imply anything about the ordering of neighbours within a processors list of neighbours. This means it is not possible to traverse the ring without some additional information.

Another way to look at this is that each processor has its own idea of whether clockwise is from its first neighbour to its second neighbour, or from the second neighbour to its first neighbour. One processor,  $a$ , may consider another processor,  $b$ , to be clockwise to itself, while at the same time  $b$  may consider  $a$  to be clockwise to itself.

There are two possible ways that all the processors can be consistent with each other. Continuing the analogy, these correspond to the two possible externally globally imposed choices of what direction clockwise is. These two choices are the two possible orientations. If all of the processors agree with one of these orientations, the ring is said to be oriented.



A processor also has an orientation. A processor's orientation is the externally imposable orientation with which the processor is in agreement. That is, they would agree on which neighbour is clockwise. A set of processors are consistently oriented if they have the same orientation. So another way to say that a ring is oriented is that all processors are consistently oriented.

The problem of ring orientation is to set the orientation of the processors so that the system is oriented. A protocol to do this for a ring of  $n$  uniform anonymous processors is as follows.

The system is a ring of  $n$  anonymous processors with identifiers  $\{1, 2, \dots, n\}$ . The set of neighbours is  $\{\{1, 2\}, \{2, 3\}, \dots, \{n-1, n\}, \{n, 1\}\}$ . *maxval* is some arbitrary finite value such that  $\text{maxval} > 2 * \text{maxnoise}$ . The value  $2 * \text{maxnoise}$  is chosen so that a processor that is close to zero or *maxval* can not at the same time be close to *maxval* or 0 respectively.

The processors are uniform, so only one processor needs to be described. The order of the list of the processor's neighbours is arbitrary. It has one variable, representing an arrow pointing to the neighbour it considers to be clockwise. If the value is negative, it points to its first neighbour, and if it positive, it points to its second neighbour. If the variable is exactly zero, it points to neither neighbour. This means that a processor is not oriented if the value of its variable is currently zero.

There is also one pseudo variable which is the negation of the processor's variable. This value is the only one exported to the first neighbour, and the value of the processor's variable is the only thing exported to the second neighbour. This is done because the values from the neighbours are to be interpreted as an arrow pointing towards or away from the processor, as opposed to telling whether the neighbour is pointing to their first or second neighbour. If the processor receives a positive value from a neighbour, that neighbour is pointing towards it. If it is negative, that neighbour is pointing away from it. If it



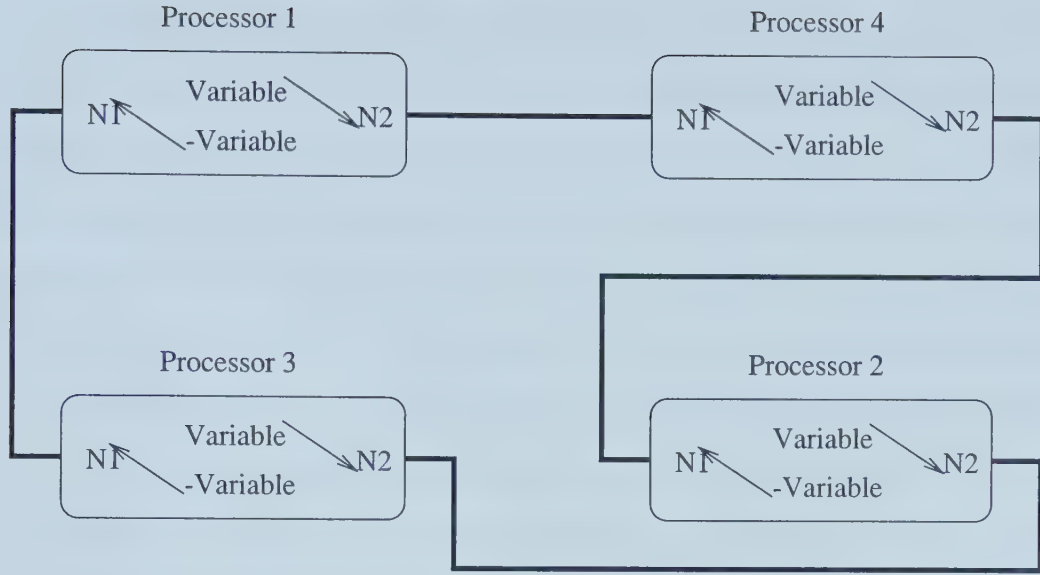


Figure 5.1: A ring of four processors.

is zero, the neighbour is neither facing towards or away from the processor.

Figure 5.1 shows a ring with four processors. One way for this ring to be oriented is if processors 1, 2, and 4 are positive, and processor 3 is negative.

In order for the given protocol to work, there are some properties that are necessary for the noise to have. First, each random process must have the Markov property, that is, future noise values do not depend on previous values. This is used in the proof to show that the probability of the system stabilising approaches 1 as time increases. Second, for each variable it must hold that for any open interval within  $(-maxnoise, maxnoise)$  there is a non zero probability of the value of the noise remaining within that interval for any length of time. Also, for any interval of time  $[t_0, t)$  it must be possible for the noise to have any value between  $(-maxnoise, maxnoise)$  at time  $t$ . Each noise function, as noted in the description of the model, is an independent random source.

A variable that is close to zero has a useful property. If it is close to zero, by definition, the stored value of the variable is less than  $maxnoise$  away from





zero. This means that in the possible range of net values for the variable, there will be an open interval where it is positive, and an open interval where it is negative. So there is a non-zero probability of the net value of the variable being either positive or negative. In order to say that the infimum of this probability over all possible states is non-zero, we will further reduce the size of this interval by some  $\epsilon$ . A processor where the magnitude of the stored value of the variable is less than  $maxnoise - \epsilon$  will be called *weakly oriented*. If it is not weakly oriented then the processor is *strongly oriented*.

Let the value of the processor's variable be  $s$ , the value of the first neighbours variable be  $a$ , and the value of the second neighbours variable be  $b$ . Let  $slowrate$  and  $fastrate$  be real values defined such that  $0 < slowrate < slowrate * (2 * maxspeedup^2 + maxspeedup) < fastrate < maxval$ . This polynomial in  $maxspeedup$  is necessary to resolve the situation shown in figure 6.6 in the proof of correctness.

Given these values, the transition function  $f$  for a processor's variable is:

$$f(s, a, b) = \begin{cases} 0 & a < 0, b < 0 \\ -slowrate & a < 0, b > 0 \\ slowrate & a > 0, b < 0 \\ fastrate & s < 0, a > 0, b > 0 \\ -fastrate & s > 0, a > 0, b > 0 \end{cases}$$

The target states in this system are the states where all the processors are close to  $maxval$ , and they all have the same orientation. That is, the system is oriented, and the processors are all as strongly oriented as is possible, or close to it.



# Chapter 6

## Correctness of Protocol

Chapter 5 defined the protocol, but did not give any argument for correctness. We give an argument of correctness here. Due to the length, we first give an overview of the proof. Then we show that a target can be reached from any state. Finally we use this to show that the system stabilises to a target state.

Because in this system, a processor has only one variable, for the sake of brevity we will sometimes refer to the processor's value rather than the value of the processor's variable.

We will also refer to chains of processors. A chain of processors is a subsection of the ring. That is, it is zero or more contiguous processors that form an arc of the ring. So, for example, in the system depicted in figure 5.1, one possible chain is processor 3, processor 1, and processor 4.

In diagrams of chains of processors, each processor is represented as an arrow which points to the neighbour the processor is pointing towards. A strongly oriented processor points exactly left or right, while a weakly oriented processor points upwards at an angle. A smaller arrow above the large arrow indicates the direction it is moving.

Consider again the system depicted in figure 5.1. Let processor 3 be strongly oriented and negative. Because its first neighbour is processor 1, processor 3 is pointing at processor 1. Let processor 1 be weakly oriented and





Figure 6.1: A chain of processors.

positive. Because its second neighbour is processor 4, processor 1 is pointing at processor 4. Let processor 4 be strongly oriented and negative. Because its first neighbour is processor 1, processor 4 is pointing at processor 1. This is depicted by figure 6.1. It is possible to draw a simplified diagram like this because the transition function is symmetric in the sense that the behaviour of a processor is preserved if the order of its neighbours is reversed and its value is negated.

There is also a slightly odd usage of the term worst case in this proof. It quickly becomes clear that in the worst case, the system will not stabilise. There is only an increasing probability that it will do so. However, we are still interested in the lowest probabilities or longest times given all the possible system states and scheduler actions. This is the worst case that we will speak of.

## 6.1 Overview

The idea of this proof is to use the noise as a “friend” against the scheduler daemon adversary. Because of the properties of the noise, for any range of values of noise that would be helpful at some point, there is a non-zero probability the noise will take a value within this range. So in a state  $S_1$ , if for every scheduler action, values can be given for the noise that will drive the system to a particular state  $S_2$ , there is some non-zero (but possibly very small) probability of the system reaching  $S_2$  from  $S_1$ . Thus there is a path, with non-zero probability, from  $S_1$  to  $S_2$ .





If  $S_1$  is instead not a specific state, but any state from within a set of possible states, this would mean that there is a non-zero probability of reaching  $S_2$  from any of the states in the set.  $S_2$  can also be extended to a set of possible states, so that from any state in the set  $S_1$  the system will reach one of the states in  $S_2$ . This is a major part of the proof in this chapter. The set of all possible states is divided up into a number of cases, and it is shown that there is a non-zero probability of reaching one of the target stable states for each of the cases.

Finally, we need an argument to say that from any state, the probability of reaching a target state can be made arbitrarily close to 1 by running the system for long enough. Because the stable states are shown to be the target states, the system is expected to reach a target state, and remain there, so it is self stabilising.

In these proofs, it is necessary to know that the target states are stable states. This is proved later on, along with the fact that all stable states are target states.

## 6.2 Reachability of Target

**Theorem 6.2.1** *If all the strongly oriented processors have the same orientation  $O$ , the system can reach a target state.*

*Proof.* There are three cases.

The first case is that there are no strongly oriented processors. In this case, let  $O$  be either of the two possible orientations, and proceed with the argument in case three.

The second case is that all the processors are strongly oriented with orientation  $O$ . By the definition of the transition function, the stored value of each processor will move toward having maximum magnitude with orientation  $O$ .



Because the processors are strongly oriented, in worst case the noise only need stay within the region  $(-maxnoise + \epsilon, maxnoise - \epsilon)$ . This means that if all the processors are strongly and consistently oriented, the system can reach a target state. Given that the noise does this, in worst case, the system will stabilise within time  $\frac{maxval - (maxnoise - \epsilon)}{slowrate}$ .

The third case is that there is at least one weakly oriented processor and one strongly oriented processor. In this case, for every weakly oriented processor, there is some interval of values of width at least  $\epsilon$  for the noise such that the processor will have orientation  $O$ . If the noise stays in this interval for all weakly oriented processors, and the noise for the strongly oriented processor does not change its orientation, all processors will tend towards  $O$ , as stated in the previous case. Thus there is a non-zero probability that the system will reach a target state. If the system follows this path, it will stabilise in worst case within time  $\frac{maxval + maxnoise + \epsilon}{slowrate}$  because a processor will have to change by at most  $maxval + maxnoise + \epsilon$  and the processors will be changing rate at least as fast as  $slowrate$ . ■

To prove anything about the remaining case will require proving some properties of chains of zero or more weakly oriented processors with strongly oriented neighbours.

**Lemma 6.2.2** *In a weakly oriented chain of processors with strongly oriented neighbours  $a$  and  $b$ , if  $a$  and  $b$  have different orientations and face towards the chain, there is a non-zero probability that the chain of weakly oriented processors remain weakly oriented and  $a$  or  $b$  becomes weakly oriented.*

*Proof.* There are three cases in this proof.

*Case 1.* There are no processors in the chain. This looks like figure 6.2. There is a chance that noise will not change  $a$  or  $b$ 's orientation and that both  $a$  and  $b$  will decrease in magnitude. Note that if  $a$  or  $b$  is not close to zero, that





Figure 6.2: Case 1.

probability is 1. The processors change at either the slow or the fast rate, depending on their neighbours. If this occurs, eventually one of  $a$  or  $b$  must become weakly oriented. In worst case, this will take  $\frac{\maxval - \maxnoise + \epsilon}{\text{slowrate}}$  time.



Figure 6.3: Case 2.

*Case 2.* There is only one processor in the chain. An example of this is shown in figure 6.3. There is a chance that the random noise is such that the weakly oriented processor spends roughly half its time in both orientations, and the strongly oriented processors do not change their orientation. So depending on their other neighbour, both  $a$  and  $b$  either spend half the time decreasing at the fast rate and half the time increasing at the slow rate, or half the time decreasing at the slow rate and half the time not changing. Because the fast rate is faster than the slow rate no matter what the scheduler does, both of these possibilities lead to a decrease in magnitude over time. So eventually  $a$  or  $b$  will have a small enough magnitude that they are weakly oriented. In worst case, this will happen by time  $\frac{\maxval - \maxnoise + \epsilon}{\max\{\text{slowrate}/2, (\text{fastrate} + \text{slowrate})/2\}}$ .

*Case 3.* There is more than one processor in the chain. In this case, there is a chance that the weakly oriented processors will be oriented as follows, and that noise does not change the orientation of the strongly oriented processors. If there are an odd number of weakly oriented processors, the processor adjacent





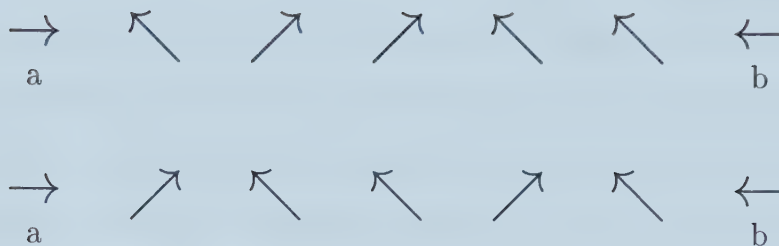


Figure 6.4: Case 3 - special configurations.

to  $b$  faces away from  $b$ . The remaining processors are divided into pairs. For each of the pairs, either both processors face each other, or face away from each other. Finally, adjacent pairs of processors are of alternating type, so that a pair of processors facing each other is beside a pair of processors facing away from each other. Note that there are two possible sets of orientation that satisfy this, and that one can be obtained from the other by changing the orientation of every paired weakly oriented processor. Call these special configurations. The two special configurations for a chain of five processors is shown in figure 6.4.

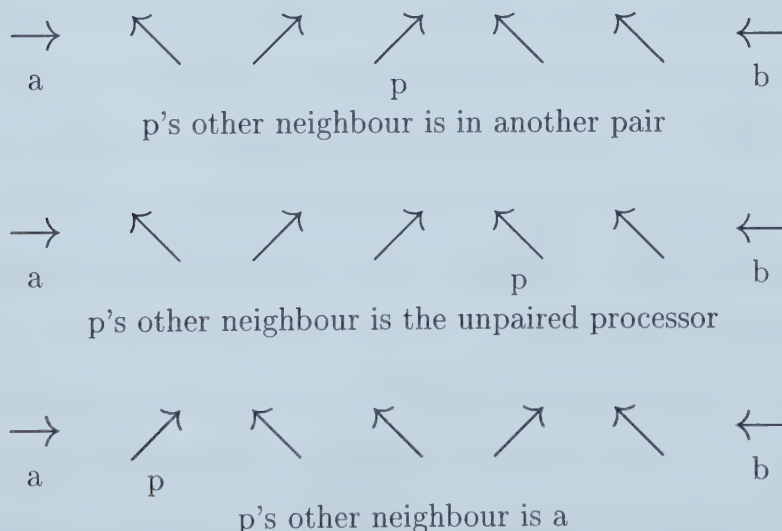


Figure 6.5: Three cases for a facing pair of processors.



Given a chain of processors in one of the two special configurations, consider a pair of processors that are facing each other. Consider a processor  $p$  in this pair.  $p$  must have at least one neighbour facing it: the other processor in the pair. Now look at the other neighbour, the one that is not in this pair of processors. This other neighbour is either  $a$  or  $b$ , a processor in another pair, or the unpaired processor. These three cases are shown in figure 6.5. The unpaired processor,  $a$ , and  $b$  always face inwards toward the chain of weakly oriented processors, so if they are  $p$ 's other neighbour, they must face  $p$ . By construction any pair of processors beside a pair that faces each other must face away from each other, and towards the neighbouring pair. So if  $p$ 's neighbour is in another processor pair, it would face towards  $p$ . Thus  $p$  has two neighbours facing towards it in all cases. This means that  $p$  heads towards zero, and away from the other processor in the pair, at the fast rate. Because  $p$  was arbitrarily chosen, both processors in the pair do this. Note that they will not overshoot zero because the processors are heading towards zero, as opposed to specifically heading away from the other processor.

Consider a pair of processors that are facing away from each other. Because each processor has at least the other processor in the pair facing away from it, both processors either move toward the other processor in the pair at the slow rate or do not change, depending on the orientation of their other neighbour.

Now consider the unpaired processor. Call it  $p$ . Because there is more than one weakly oriented processor, it must be adjacent to  $b$  and a pair of weakly oriented processors. When the processors in the pair face each other then  $p$  moves away from  $b$  at the slow rate. When the pair face away from each other then  $p$  heads to zero, which is towards  $b$ , at the fast rate.

In a pair of processors  $x$  and  $y$  that face each other,  $x$  may reach zero before  $y$ , at which point  $y$  will stop moving. This is shown in figure 6.6. However, once  $x$  reaches zero there is a chance that random noise will repeatedly change





Figure 6.6:  $x$  has reached zero before  $y$ .

$x$ 's orientation. Because  $x$  moves in a different direction in each orientation, to keep  $x$  roughly stationary it must spend enough time in one orientation to counteract the movement made in the other orientation. Due to scheduler intervention, this would mean  $x$  may spend up to  $maxspeedup$  times more time in one orientation than the other. While  $x$  is facing  $y$ ,  $y$  will head away from  $x$  at the fast rate, and while  $x$  is facing away from  $y$ ,  $y$  will head towards  $x$  at the slow rate. Because of the way that the fast rate is defined this means that  $y$  will still move away from  $x$  faster than  $slow\ rate * maxspeedup$ . This is, in fact, the reason behind the definition of the fast and slow rate.

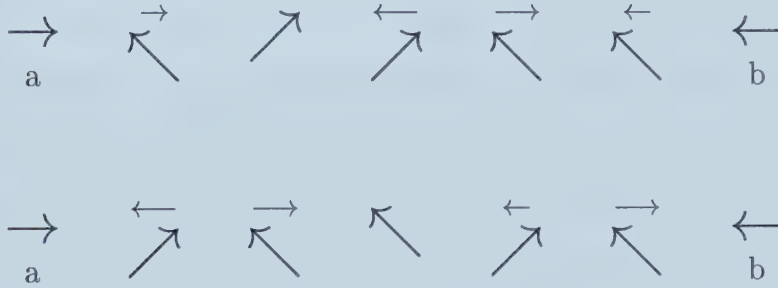


Figure 6.7: Case 3 - alternating special configurations cancel movement of weakly oriented processors.

Running in one of the special configurations can counteract the movement of the processors in the other special configuration, and by doing so keep the processors weakly oriented. Consider a pair of processors. In the special configuration where they face each other, both processors will move away from each other faster than  $slow\ rate * maxspeedup$ , but will not pass zero because





the processor is moving towards zero at the fast rate, not in a particular direction. In the other special configuration, each processor will either stay stationary, or head towards the other processor in the pair no faster than  $slow\ rate * maxspeedup$ . Spending roughly the same amount in each special configuration will keep the processors close to zero, and thus weakly oriented. Now consider the unpaired processor. It alternates between moving away from  $b$  no faster than  $slow\ rate * maxspeedup$  and moving towards  $b$  at least as fast as  $fast\ rate$ , but again, not past zero. This is shown in figure 6.7. Spending roughly the same amount of time in each special configuration will keep this processor weakly oriented as well. So if the chain spends roughly half its time in each special configuration, changing before any slowly moving processor changes orientation, all the weakly oriented processors will remain weakly oriented.

Now consider what happens to  $a$  if the chain follows the behaviour above, and it does not change its orientation due to noise. It must have a pair of weakly oriented processors beside it, because there is more than one weakly oriented processor. Let  $p$  be the processor in this pair that is beside  $a$ . In the special configuration with  $p$  facing away from the other processor in the pair,  $p$  moves away from  $a$  at the slow rate. This means that  $p$  will face  $a$  for the entire time the chain is in this special configuration. In the other special configuration,  $p$  will move towards  $a$  at the fast rate, up to zero, counteracting the movement it made in the other special configuration. This means that for about half the time  $a$  will have at least  $p$  facing it, and at most half the time  $p$  will face away from  $a$ . If  $a$ 's other neighbour faces  $a$ ,  $a$  will spend half its time decreasing at the fast rate and half the time increasing at the slow rate, and if it faces away from  $a$ ,  $a$  will spend half the time decreasing at the slow rate and staying the same the other half. In either case,  $a$  decreases over time.

Thus following this behaviour, if  $b$  does not become weakly oriented at



some point on its own,  $a$  will become weakly oriented given enough time. In either case, one of  $a$  or  $b$  has become weakly oriented, and all weakly oriented processors have remained weakly oriented. Despite the extra complication, in the worst case, this will still happen by time  $\frac{maxval - maxnoise + \epsilon}{\max\{slowrate/2, (fastrate + slowrate)/2\}}$ . ■

**Lemma 6.2.3** *In a weakly oriented chain of processors with neighbours  $a$  and  $b$ , where  $a$  and  $b$  have the same orientation  $O$ , there is a non-zero probability that the chain of weakly oriented processors become strongly oriented with orientation  $O$  and both  $a$  and  $b$  remain strongly oriented with orientation  $O$ .  $a$  and  $b$  may be the same processor, when the chain consists of the entire ring.*

*Proof.* Similar to the proof of theorem 6.2.1, random noise may be such that all the weakly oriented processors have orientation  $O$ , and remain that way long enough that they become strongly oriented, and the strongly oriented processors do not change orientation. Also like in theorem 6.2.1, in worst case this will occur by time  $\frac{maxval + maxnoise + \epsilon}{slowrate}$ . ■

**Lemma 6.2.4** *In a weakly oriented chain of processors with neighbours  $a$  and  $b$ , if  $a$  and  $b$  have different orientations and face away from the chain, there is a non-zero probability that  $a$  and  $b$  remain strongly oriented with the same orientation, and no new chains of similarly strongly oriented processors will be created.*

*Proof.* If there are no weakly oriented processors in the chain, all that needs to happen is for the strongly oriented processors to not change orientation. If they are not close to zero, this happens with probability 1.

Otherwise, there is a chance that all the weakly oriented processors will have the same orientation as  $a$ , and will remain that way long enough for all but the processor adjacent to  $b$  to become strongly oriented. Also, the



strongly oriented processors may not change orientation. If this occurs, the processor adjacent to  $b$  might not become strongly oriented, because all the adjacent processors face away from it and are strongly oriented, and possibly unable to change orientation. If there is only one weakly oriented processor in the chain, this is already the case. This does not create any new chains of similarly strongly oriented processors, it merely extends the length of the existing chain that contains  $a$  towards  $b$ . In worst case, this will occur by time  $\frac{\maxval + \maxnoise + \epsilon}{\text{slowrate}}$ . ■

**Theorem 6.2.5** *If there are at least two strongly oriented processors, with different orientations, there is a non-zero chance that the system will reach a target state.*

*Proof.* Divide the system up into consistently oriented chains of strongly oriented processors of maximal size. There is a probability that each of the processors in the interior of a chain of at least three processors does not change orientation due to noise, and so they simply increase in magnitude, and are not of further interest. At the junction between these chains, one of lemmas 6.2.2, 6.2.3, or 6.2.4 can be applied. Note that the behaviours in these lemmas do not interfere with the behaviours of the lemmas in any other locations.

Consider what happens after the application of one these lemmas. If lemma 6.2.3 is applied, the string of weakly oriented processors the lemma is applied to will be gone, and the two strongly oriented chains adjacent to it will be merged. If lemma 6.2.4 is applied, one of the adjacent strongly oriented chains will be lengthened, but the chains will continue to be separated by one weakly oriented processor. Finally, if lemma 6.2.2 is applied, one of the adjacent strongly oriented chains will be shortened by one processor, which becomes part of the weakly oriented chain. After at most  $n - 1$  applications of this lemma ( $n - 1$  is the length of the longest possible chain given that there must



be at least two separate chains) one of the adjacent strongly oriented chains will be gone, completely subsumed by the weakly oriented chain.

Note that chains of strongly oriented processors that point towards each other are not introduced by lemmas 6.2.3 and 6.2.4, and lemma 6.2.2 will remove one such chain. Because lemma 6.2.2 can be applied as long as there are two strongly oriented chains with different orientations, and there can be at most  $n$  chains, it will take no longer than  $n * (n - 1)$  applications of lemma 6.2.2 for all strongly oriented chains to be similarly oriented. So if this set of behaviours is run continuously, there will eventually be no two strongly oriented chains with different orientations. At this point, by theorem 6.2.1, there is a non-zero chance that system will now reach a target state. As a final note, while the function is not clear, the worst case probability is a function of  $\epsilon$ ,  $n$ , and the constants *maxval*, *fastrate*, and *slowrate*. So for fixed  $n$  and  $\epsilon$  the infimum of the probabilities over the worst case for all possible states and times is not zero. ■

## 6.3 Proof of Stabilisation

**Theorem 6.3.1** *The stable states in the system are exactly the target states.*

*Proof.* Assume the system is close to a target state  $S$ . Then the stored values of all processors must be greater than  $\text{maxval} - \text{maxnoise}$ , and thus they must be strongly oriented because  $\text{maxval} > 2 * \text{maxnoise}$ . Because all the processors agree on the orientation, and are strongly oriented, the magnitudes of the stored values can only increase, no matter what the scheduler or random noise does. But the magnitude can not increase beyond  $\text{maxval}$ , so the magnitude of the net value will always be within  $\text{maxnoise}$  of the value in  $S$ , which is itself close to  $\text{maxval}$ . Thus  $S$  is a stable state.

Assume  $S$  is a stable state. By theorem 6.2.5, there is a path with non-





zero probability leading to a target state. So  $S$  must be close to some target state. Because  $\text{maxval} > 2 * \text{maxnoise}$ , in state  $S$  all the processors must be consistently strongly oriented. By the argument in theorem 6.2.1, this means that their magnitude will increase until it reaches  $\text{maxval}$ . In order for  $S$  to be stable, all processors must have started close to  $\pm \text{maxval}$ . The only way for  $S$  to be close to both a target state and have all processors close to  $\pm \text{maxval}$  is for  $S$  to be a target state. ■

**Theorem 6.3.2** *The probability that the system has stabilised goes to 1 as time goes to infinity.*

*Proof.* Consider a system in a state  $S$  at some time  $t_0$ . Let  $f_{i,j}(T)$  describe the noise applied to the  $i$ 'th processor's  $j$ 'th variable.  $S$  includes the values of the noise, so  $f_{i,j}(t_0)$  must be equal to the values specified in  $S$ . By theorems 6.2.1 and 6.2.5, we know there is some  $\epsilon' > 0$  and  $t > 0$  such that the probability of the system reaching a target state some time between  $t_0$  and  $t_0 + t$  is greater than  $\epsilon'$ . By theorem 6.3.1, we know that the system will not leave the region around a target state, so the probability of being close to a target state at time  $t$  must also be at least  $\epsilon'$ .

Because the future values of the system are completely determined by the current value of the system, and the values of the noise, the probability above is the probability of the functions  $f_{i,j}(T)$  having certain values. Because the noise has the Markov property, the probability of the noise being described by some set of functions with the same value at  $t_0$  is not affected by the history of the system. This allows us to say that probability of not reaching a target state for  $i$  intervals of length  $t$  is less than  $(1 - \epsilon')^i$ . This goes to 0 as  $i$  goes to infinity, so as time increases, the probability of the system stabilising goes to 1. ■



# Chapter 7

## Hardware Implementation

In the previous chapters, we make a number of references to the actual hardware that we built. In this chapter we give more detail about what we did with the hardware we had, and the problems we had dealing with it.

All the hardware we built was based on the AD713 quad operational amplifier and the AD633 analog multiplier, with a  $\pm 15\text{V}$  power supply. We had enough components to make eight processors, with at most three processors on a single breadboard. Data sampling was done with an ADAC 5500MF data acquisition board at approximately 25000 samples a second, distributed over however many channels were being monitored. While it is limited to  $\pm 10\text{V}$ , we were not concerned because the interesting changes occur closer to 0, and values with magnitude greater than are simply clipped to 10.

The transition function given in chapter 5 is not directly implementable in hardware. If the processors own value is  $s$ , and its first and second neighbour's values are  $a$  and  $b$  respectively, the corresponding hardware function is  $(\text{sign}(a) - \text{sign}(b))/10 - \text{sign}(s) * (\text{sign}(a) + \text{maxval}) * (\text{sign}(b) + \text{maxval})$ . This would make  $\text{fastrate}$  equal to  $10 * \text{slowrate}$ . This circuit is shown in 7.1.

There is an additional difference between the hardware implementation and the theoretical protocol. The circuit uses both  $\text{sign}(a)$  and  $-\text{sign}(a)$  to compute the function, and both  $\text{sign}(s)$  and  $-\text{sign}(s)$  are generated by the



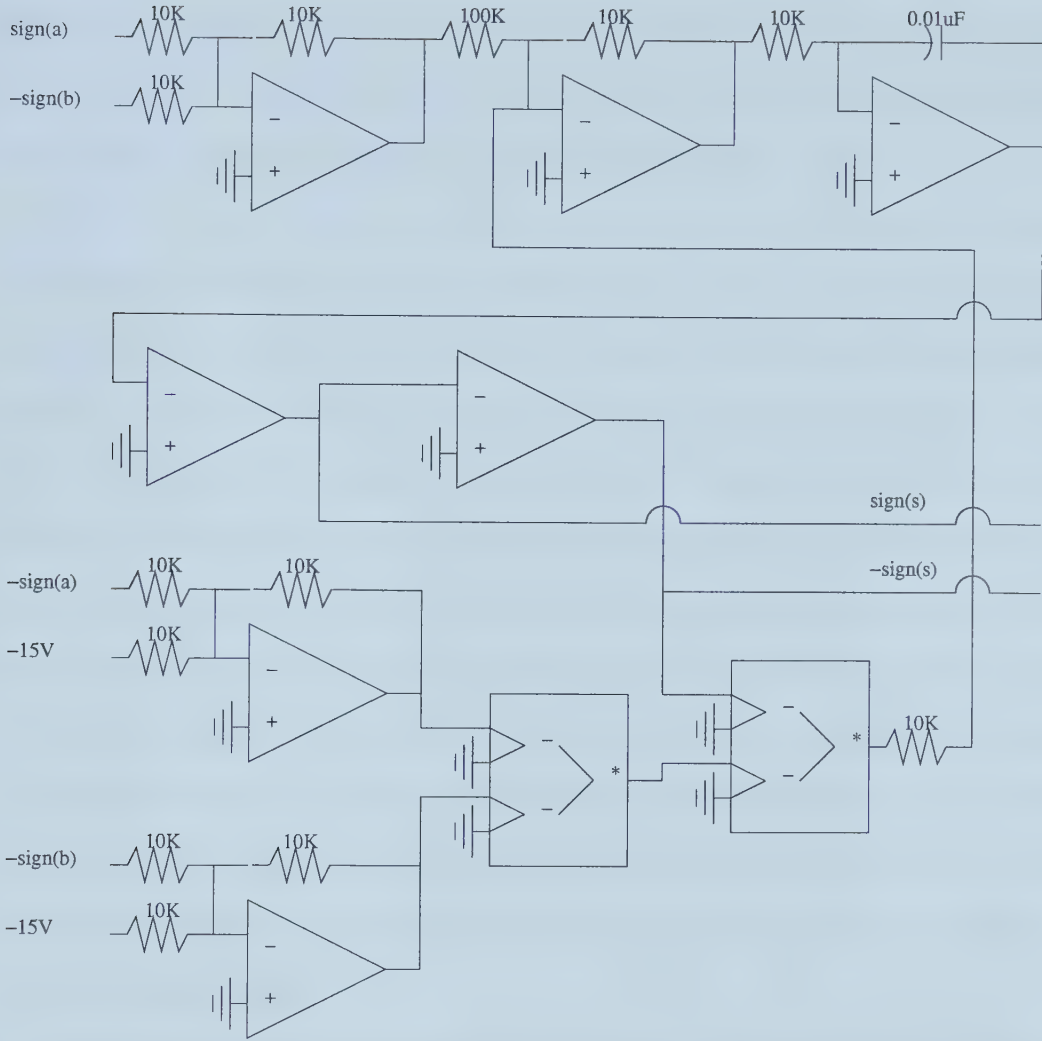


Figure 7.1: Circuit diagram for the ring orientation protocol

processor. Because of this, it was easy to pass both the sign of the exported value and the sign of its negation is passed to its neighbours. This means that a processor passes  $-\text{sign}(s)$  to  $a$  as the exported value and  $\text{sign}(s)$  as its negation, and it passes  $\text{sign}(s)$  to  $b$  as the exported value and  $-\text{sign}(s)$  as its negation.

One thing that we noted while working with the hardware is that the size of the capacitors used in the integration feedback loops can not be too large. The capacitors that we normally used were  $0.01\mu\text{F}$  capacitors. While testing individual processors we slowed things down by using much larger  $22\mu\text{F}$





capacitors, and everything appeared to work normally. However, when we had the whole system together, and we tried the same trick, the system would no longer work. One or two processors would end up stuck at 0V.

While this would appear to invalidate the proof given in chapter 6, there is a reasonable explanation for this, and this was predicted. When a processor is close to zero, in order for it to become strongly oriented, the random noise must stay above (or below) a certain value for a long enough period of time that the processor is no longer weakly oriented. Since we were relying on naturally occurring noise, slowing down the processors also had the effect of speeding up the noise. By choosing large enough capacitors, the noise may have been changing values too quickly for this to occur. The fact that the system did eventually stabilise one time (after being left on overnight after leaving in frustration) could be taken as corroborating evidence for this. This could possibly be further tested by artificially injecting better controlled noise into the system. This would also help ensure that the noise would be independent between the processors.

A problem that we were not able to solve to our satisfaction with the hardware available to us was monitoring the system. This was because the processors we built stabilised too quickly for us to watch. While it was clear that the system stabilised, we could not get much data telling us how it stabilised. Initially, we tried watching the system as it powered on. This provided no information, as all pertinent data was washed out in the initial fluctuations of the power supply. Next, we tried letting the system settle, and then forcing one of the processors to have the opposite orientation. Data from a sample run is shown in table 7.1.

This data is from a ring of six processors where each processor  $x$ 's first neighbour is  $(x+1 \text{ modulo } 6)+1$ , and its second neighbour is  $(x-1 \text{ modulo } 6)+1$ . The first column of data is from processor 1, the second from processor 2,



+10.006	+10.006	+10.006	+10.006	+10.006	+10.006
+10.006	+10.006	+10.006	+10.006	+10.006	+10.006
+10.006	+10.006	+10.006	+10.006	+10.006	+10.006
+10.006	+10.006	+10.006	+10.006	+10.006	+10.006
+9.933	+10.006	+10.006	+10.006	+10.006	-8.901560
-10.011	+10.006	+10.006	+10.006	-9.547	+10.006
-10.011	+10.006	+10.006	-8.765	-0.005	-8.838
-10.011	+10.006	-7.513	+3.129	-9.703	-8.765
-10.011	+10.006	+10.006	+5.001	+10.006	-8.765
-10.011	+10.006	+4.375	-9.977	-10.006	-9.390
-10.011	+10.006	+10.006	+9.693	-9.977	-4.385
-10.011	+10.006	+10.006	-5.010	+7.347	+4.219
-10.011	+10.006	+10.006	+10.006	+10.006	-9.390
-10.011	+10.006	+10.006	+10.006	-9.390	+10.006
-10.011	+10.006	-9.997	-8.129	+10.006	+10.006
-10.011	-10.011	-10.011	-10.011	-10.011	-10.011
-10.011	-10.011	-10.011	-10.011	-10.011	-10.011
-10.011	-10.011	-10.011	-10.011	-10.011	-10.011
-10.011	-10.011	-10.011	-10.011	-10.011	-10.011

Table 7.1: processor one forced to be negative

etc. While the data is arranged in rows, which may suggest that each processor is sampled concurrently, each processor is in fact sampled in turn. In this run, processor 1 has been forced to change orientation by tying it to  $-maxval$ .

At the 12th row of data it is clear that something odd is going on. Processor 6, which is now being pointed to by processor 1, is quite strongly pointing towards processor 1, after having pointed away from it in row 11. This occurs again in rows 14 and 16 with processor 6, and other rows with other processors. This should not be possible, as a processor facing a processor facing back at it should never increase. The most likely explanation is that a solid contact tying processor 1 to  $-maxval$  is not made instantly, and the results are the result of “switch bounce”. Because the values of processors change so drastically between samplings, it was not possible to confirm this with our sampling hardware.

In an attempt to try and eliminate this, we then tried generating a pulse



-10.011	-10.011	-10.011	-10.011	-10.011	-10.011
-10.011	-10.011	-10.011	-10.011	-10.011	-10.011
-10.011	-10.011	-10.011	-10.011	-10.011	-10.011
-10.011	-10.011	-10.011	-10.011	-10.011	-10.011
-10.011	+10.006	+10.006	+10.006	+10.006	+10.006
+10.006	+10.006	+10.006	+10.006	+10.006	+10.006
+10.006	+10.006	+10.006	+10.006	+10.006	+10.006
+10.006	+10.006	+10.006	+10.006	+10.006	+10.006
+10.006	+10.006	+10.006	+10.006	+10.006	+10.006

Table 7.2: processor one forced to be negative with a pulse

using only solid state hardware. We adjusted a pulse generator until the pulse was around 20 samples long. This corresponds to about 3 rows when sampling six processors. The data from this run is shown in table 7.2. The fact that the system stabilises so cleanly here would seem to indicate that the problems with the method used above was indeed a noisy connection, but it is still clear that a much higher sampling rate is needed to get useful information.



# Chapter 8

## Average Time, Better Bounds, and Simulation

This chapter details the incomplete work done to try and determine the average and worst case behaviour of the ring orientation protocol. This worst case is of the same sense as in the proof: it is something like an average case under certain adverse conditions. While a proof of stabilisation is nice, simply knowing that the system will eventually stabilise is not very satisfying. We would like to know something about how quickly it stabilises.

The first thing we thought of doing was using the correctness proof to get an idea of the worst case behaviour. Given some work, an exact function for the worst case probability for the scenario outlined in the proof of correctness could be determined and this could be used to give an upper bound on the worst case. It would only be an upper bound, and not an exact function, because doing this would only take into account one possible route to a target state, not all possible routes. However, it seems fairly clear that with the argument as given, this would be some sort of exponential function of  $n$ . This function was not determined because it did not seem like a good investment of time to exactly determine the value of an exponential upper bound.

Unfortunately, the only other idea that we had would be to look at the probabilistic system of differential equations controlling the derivative, and





	S is 0		S is 1	
	A is 0	A is 1	A is 0	A is 1
B is 0	1	0	1 or 2	0 or 1 or 2
B is 1	1	0	1 or 2	1 or 2
	S is 2		S is 3	
	A is 0	A is 1	A is 0	A is 1
B is 0	1 or 2	1 or 2	2	2
B is 1	1 or 2 or 3	1 or 2	3	3

Table 8.1: transition function for discrete processor

exactly determine the probability of being in a given state after some length of time. Unfortunately, this would involve solving a system of  $n$  non-continuous differential equations in  $n$  variables. Not only were we missing the mathematical skill to do this, it is certainly not a given that a closed form solution even exists, and it was not even clear how to add the scheduler effects into the equations. Given this, we decided to try looking at a simplified version of the problem. The simplified protocol that we looked at was the following discrete system.

The system is a ring of  $n$  processors. Each processor can be in one four states: 0 (strongly towards the first processor), 1 (weakly towards the first processor), 2 (weakly towards the second processor) or 3 (strongly towards the second processor.) It passes one of two values to each of its neighbours: 0 (towards) or 1 (away). If the current value of the processor is  $S$ , the value from the first processors is  $A$ , and the value from the second processor is  $B$ , the transition function giving the next state is shown in table 8.1. If there is more than one choice, the next value is chosen randomly with equal probability for each choice.

Because we did not yet have an argument showing correctness, we were also interested in the correctness of this system. To prove correctness, we wrote



a program that systematically explores the whole state space. This program computes  $P_{i,S}(A)$ , the probability of a system starting in state  $S$  avoiding a target state for  $i$  steps. Because the probabilities at each step are independent,  $P_{i,S}(A) = \sum_{S' \in S} P_S(S') * P_{i-1,S'}(A)$ , where  $P_S(S')$  is the probability that a system in state  $S$  will be in state  $S'$  in the next time step. The scheduler is added by restricting the next system states to the ones that the scheduler would choose. A smart scheduler would choose the set of choices that maximises these probabilities.

The program first computes  $P_{0,S}(A)$  for each state, which is one if  $S$  is not a target state, and zero if it is. Then it continues to iterate through the steps, computing the new  $P_{i,S}(A)$  for each state. If after some number of iterations,  $P_{i,S}(A)$  is less than one for all states  $S$ , then the system will eventually stabilise. Using this idea, we were able to show correctness when all processors are activated at every step. If the scheduler can activate any subset of processors for an update (a distributed scheduling demon) the protocol fails, unless there is some finite upper bound on the number of steps for which a processor can be idle.

The average number of steps to stabilise is  $\sum_{i=0}^{\infty} P_{i,S}(A) * i$ . Because the probabilities of not stabilising go to zero, if enough steps of the checker are run, this value can be approximated by keeping a running sum of  $P_{i,S}(A) * i$  and stopping when the values of  $P_{i,S}$  get small enough. Our program defined small to be less than 0.000000001. For simplicity and speed, we used the scheduler that updates every processor at each step. Doing this, we were able to compute the values of the average time until stabilisation for 1 to 10 processors. These values are given in table 8.2 and graphed in figure 8.1. The close correspondence to  $5 * (n - 1) * n$  is very striking.



number of processors	number of steps
1	1.500000
2	10.000000
3	29.300758
4	58.381769
5	95.253113
6	142.462688
7	201.546215
8	274.251274
9	362.572740
10	468.810007

Table 8.2: average number of steps to stabilise

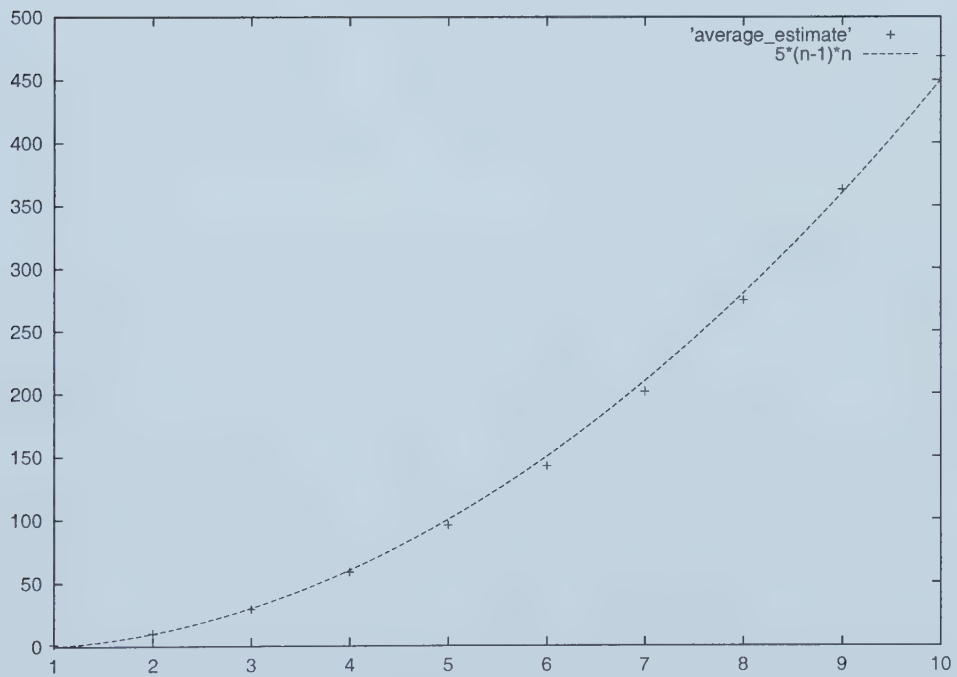


Figure 8.1: average number of steps to stabilise





# Chapter 9

## Conclusions

In the previous chapters, we accomplish two main things. The first is a definition of a hardware model for distributed continuous computation. The second is a continuous self stabilising protocol for ring orientation.

The model that we define is a refinement of the ideas in [4]. It considers more details necessary for a formal model. It also allows the control function to be piecewise continuous, which is useful in decision making. Finally, there are a few modifications made in an attempt to decrease model complexity. We believe that this model is easier to work with, without sacrificing power or accuracy.

The protocol we give for ring orientation was constructed within the model that we defined. One useful effect of doing this is that it demonstrates that the model is usable. In addition to the definition of the protocol, we give a proof of correctness within the model. This is externally validated by the fact that an actual hardware implementation of the protocol works.

There are a number of possible future areas of research. One would be to compute the time for stabilisation of the ring orientation protocol. While some work was done in this direction, it did not produce usable results.

Another question is whether it is possible to have the same computational power if the model only allows functions consisting of an unbounded finite



number of regions, each of which has some constant value throughout the region. While we could not implement our protocol if the regions could only take one of three possible values, we did not show anything about a model that can use a finite unbounded number of different values.

There are certain properties needed for the noise in the argument about correctness. It is not clear what the minimum set of assumptions would be. Is it possible to remove some of these properties and have the protocol remain correct? This also leads to the question of how the properties of the noise affects the computational power of the model.

There is also the possibility of examining any of the self stabilising protocols that use a discrete computational model and creating an analog counterpart. Given enough examples of analog versions of discrete protocols, it would be interesting to try and make some sort of generalisation about the resulting protocol, the process of conversion, and even the stabilisation time.



# Bibliography

- [1] Manuel Lameiras Campagnolo, Christopher Moore, and Jose Felix Costa. Iteration, inequalities, and differentiability in analog computers. *Santa-Fe Institute working paper*, 99-07-043, 1999.
- [2] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
- [3] H. James Hoover. Feasible real functions and arithmetic circuits. *SICOMP*, 19(1):182–204, 1990.
- [4] H. James Hoover. On the self-stabilisation of processors with continuous states. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 8.1–8.15, 1995.
- [5] H. James Hoover and Piotr Rudnicki. Uniform self-stabilising orientation of unicyclic networks under read/write atomicity. *Chicago Journal of Theoretical Computer Science*, 2(5), 1996.
- [6] Amos Israeli and Marc Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104:175–196, 1993.
- [7] James, Smith, and Welford. *Analog Computer Simulation of Engineering Systems*. Intext Educational Publishers, 1971.

















University of Alberta Library



0 1620 1264 8240

**B45408**